Generative Programming from a Domain-Specific Language Viewpoint

Charles Consel

Generative Programming from a DSL Viewpoint

Charles Consel

INRIA/LaBRI
ENSEIRB – 1, avenue du docteur Albert Schweitzer,
Domaine universitaire - BP 99
33402 Talence Cedex, France
consel@labri.fr http://compose.labri.fr

1 Introduction

A domain-specific language (DSL) is typically created to model a program family [1]. The commonalities found in the target program family suggest abstractions and notations that are domain specific. In contrast with general-purpose languages (GPL), a DSL is readable for domain experts, often concise, and usually declarative. As an illustration, consider a program family aimed to communicate data in a distributed heterogeneous system. Such a layer is commonly needed in a variety of distributed applications and relies on a mechanism like the Sun Remote Procedure Call (RPC) [2]. The XDR layer of the Sun RPC consists of marshaling and un-marshaling both arguments and returned value to/from a machine-independent format. The program family, represented by all the possible (un-)marshaling variations, has lead to the development of a DSL that allows a programmer to concisely express the type of the remote procedure arguments and returned value, and to obtain the corresponding marshaling layer on both the client and server sides.

From a DSL viewpoint, generative programming [3] provides a variety of approaches and techniques to produce and optimize a DSL implementation such as the marshaling layer in the XDR case.

Outline. Section 2 discusses how generative tools can be used to compile DSL programs into GPL programs, from a DSL interpreter. When compiled into a GPL, a DSL program can be processed by existing generative tools for various purposes, including optimization, instrumentation and verification. In this context, the generative tools are driven by domain-specific information that is translated into different forms: declarations (Section 3), annotations (Section 4), and meta-programs (Section 5). In essence, these forms enable a DSL to be interfaced with existing generative tools.

2 High-Level Compilation

Some level of compilation can be achieved by specializing an interpreter with respect to a DSL program. Traditionally, this approach has been used to generate

compilers from denotational-style language definitions [4, 5]. This approach was later promoted by Consel and Marlet in the context of DSLs, where the language user base often forbids major compiler development. Furthermore, the high-level nature of DSLs facilitates the introduction of optimizations.

An example of such a compilation strategy was used for a DSL, named Plan-P, aimed to specify application-specific protocols (*e.g.,* stream-specific degradation policies) to be deployed on programmable routers [6]. Program specialization was used at run time to achieve the effect of a *Just In Time* compiler, by specializing the Plan-P interpreter with respect to a Plan-P program. The resulting compiled programs ran up to 50 times faster than their interpreted counterparts [6]. Importantly, such late compilation process enabled the safety and security of programs to be checked at the source level by the programmable routers, before being deployed. The use of specialization allowed to achieve late compilation without requiring any specific development, besides writing the interpreter.

At a lower level, meta-programming provides an alternative approach to deriving a compiler from an interpreter [7, 8]. This approach involves a careful annotation, and sometimes re-structuring, of the interpreter.

3 From a DSL Program to Declarations

A key feature of the DSL approach is to make domain-specific information an integral part of the programming paradigm. As such the programmer can be viewed as being prompted by the language to provide domain-specific information. This information may take the form of domain-specific types, syntactic constructs and notations. This language enrichment over GPLs is typically geared towards collecting sufficient information to make some domain-specific properties decidable [9] and thus to enable domain-specific verifications and optimizations. The collection of information may be achieved by dedicated program analyses, which are, by design of the DSL, simpler than the ones developed for GPLs.

Because the scope of computations to be expressed by a DSL is usually narrow, GPL constructs and operations are restricted or excluded. Furthermore this language narrowing may also be necessary to enable key properties to be statically determined. In fact, a DSL is commonly both a restricted and an enriched version of a GPL.

Once key properties are exhibited, the DSL program can be compiled into a GPL program. To retain domain-specific information, the generated program needs to be accompanied by some form of declarations specifying its properties. Of course, the declarations are tailored to a set of verification and/or optimization tools.

In the XDR case, an XDR description often defines fixed-size RPC arguments. When this description is compiled into GPL code, it can be accompanied by declarations aimed to drive some transformation tool. This situation is illustrated by the XDR compiler that conventionally generates C code gluing calls to a generic library. We have modified the XDR compiler to generate binding-time

information about the RPC argument sizes, besides the marshaling code. As a result, a declaration is attached to the marshaling code of each data item to be transmitted. This declaration defines the size parameter as static, if it corresponds to a data item that has a fixed size; it is dynamic otherwise[1]. In this work, the generated declarations are targeted for a program specializer for C, named Tempo [10]. This tool performs a number of optimizations on both the marshaling code and the generic XDR library, including the removal of buffer overflow checks and the collapsing of function layers.

The XDR case is interesting because it demonstrates that, although a DSL introduces a new programming paradigm, it can still converge with and re-use GPL technology. Additionally, because GPL tools are often used in a narrow context, the results may be more predictable. In the XDR case for instance, since the compilation schemas, the XDR library, and the specialization contexts are fixed, specialization is fully predictable. This situation obviously does not exist for an arbitrary program with an arbitrary specialization context.

Aspect-oriented declarations could also be generated from a DSL program. For example, one could imagine adding a data compression phase to marshaling methods when invoked with data greater than a given size. In this case, the pointcut language has to be expressive enough to enable any program point of interest to be used to insert the compression phase. Alternatively, annotations can be directly injected in the generated program.

4 From a DSL Program to Annotations

Instead of generating a GPL program together with declarations, annotations can be directly inserted into the generated program. This strategy allows information to be accurately placed in the program. Like declarations, these annotations are geared towards specific tools. They can either be processed at compile time or run time.

At compile time, annotations can be used to guide the compilation process towards improving code quality or code safety. In the XDR case, for example, one could imagine a library where there would be two sets of buffer operations, with or without overflow checks. The selection of an operation would depend on annotations inserted in the program.

At run time, annotations can trigger specific actions upon run-time values. For instance, a data of an unknown size could be tested before being transmitted and be compressed if it is larger than a given threshold.

5 From a DSL Program to Meta-programming

A DSL program can also benefit from meta-programming technology. In this context, the compilation of a DSL program produces a code transformer and

[1] Note that these declarations go beyond data sizes. A detailed description of the language of specialization declarations and its application to the XDR example can be found elsewhere [10, 11].

code fragments. For example, in a multi-stage language like MetaOCaml [7], the code transformer consists of language extensions that enable to concisely express program transformations.

In the XDR case, multi-stage programming would amount to generate code that expects some data size and produces code optimized for that size. Like program specialization, multi-stage programming corresponds to GPL tools that are directly used by a programmer. In the context of DSLs, these tools can implement a specific phase of an application generator.

6 Conclusion

We examined generative programming approaches and techniques from a DSL viewpoint. We showed that a DSL can make use of these approaches and techniques in very effective ways. In essence, the DSL approach exposes information about programs that can be mapped into the realm of generative programming and be translated into declarations or annotations, which would normally be provided by a programmer. This situation illustrates the high-level nature of the DSL approach.

References

1. Consel, C.: From A Program Family To A Domain-Specific Language. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 19–29
2. Sun Microsystem: NFS: Network file system protocol specification. RFC 1094, Sun Microsystem (1989)
3. Czarnecki, K., Eisenecker, U.: Generative Programming. Addison-Wesley (2000)
4. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Charleston, SC, USA, ACM Press (1993) 493–501
5. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall (1993)
6. Thibault, S., Consel, C., Muller, G.: Safe and efficient active network programming. In: 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana (1998) 135–143
7. Taha, W.: A Gentle Introduction to Multi-stage Programming. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 30 – 50
8. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. In: Domain-Specific Program Generation; International Seminar, Dagstuhl Castle. Springer-Verlag (2004) 51 – 72
9. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: Proceedings of the 10^{th} International Symposium on Programming Language Implementation and

Logic Programming. Number 1490 in Lecture Notes in Computer Science, Pisa, Italy (1998) 170–194
10. Consel, C., Lawall, J., Le Meur, A.F.: A tour of Tempo: A program specializer for the C language. Science of Computer Programming (2004)
11. Le Meur, A.F., Lawall, J., Consel, C.: Specialization scenarios: A pragmatic approach to declaring program specialization. Higher-Order and Symbolic Computation **17** (2004) 47–92